DESIGN OF FFT PROCESSOR

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Electronics and Instrumentation Engineering

BY

PAVAN KUMAR JAIN

108EI006

MD. ANISH

108EI028

Department of Electronics & Communication Engineering
National Institute of Technology
Rourkela – 769008.

MAY 2012

DESIGN OF FFT PROCESSOR

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Electronics and Instrumentation Engineering

BY

PAVAN KUMAR JAIN

108EI006

MD. ANISH

108EI028

Under the guidance of

Prof. KAMALA KANTA MAHAPATRA

Department of Electronics & Communication Engineering
National Institute of Technology
Rourkela – 769008.

MAY 2012

NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA

CERTIFICATE

This is to certify that the Thesis Report entitled "**DESIGN OF FFT PROCESSOR**" submitted by Mr. Pavan Kumar Jain and Mr. Md. Anish in partial fulfillment of the requirements for the award of Bachelor of Technology degree Electronics and Instrumentation Engineering during session 2008-2012 at National Institute of Technology, Rourkela (Deemed University) and is an authentic work by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university/institute for the award of any Degree or Diploma.

Date:14/05/2012
Place: Rourkela

Prof. **KAMALA KANTA MAHAPATRA**
Dept. of E.C.E
National Institute of Technology
Rourkela-769008

ACKNOWLEDGEMENT

First of all, we would like to express our deep sense of respect and gratitude towards our advisor and guide **Prof. Kamala Kanta Mahapatra**, who has been the guiding force behind this work. We are greatly indebted to him for his constant encouragement, invaluable advice and for propelling us further in every aspect of our academic life. His presence and optimism have provided an invaluable influence on our career and outlook for the future. We consider it our good fortune to have got an opportunity to work with such a wonderful person.

We would like to express our gratitude to all the research scholars whose works were referred to by us during the completion of this project work. A special mention about Prof. Ayas kanta Swain, Mr. Vijay Sharma, Mr. Umakanta Nanda and Mr.Jagannath Mohanty for their timely and valuable guidance that helped us to finish the work in the stipulated period of time.

We would like to thank all faculty members and staff of the Department of Electronics and Communication Engineering, N.I.T. Rourkela for their generous help in various ways for the completion of this thesis.

We would like to thank all our friends and especially our classmates for all the thoughtful and mind stimulating discussions we had, which prompted us to think beyond the obvious. We have enjoyed their companionship so much during our stay at NIT, Rourkela.

Pavan Kumar Jain
108EI006

Md. Anish
108EI028

ABSTRACT

In this project our goal is to design a processor for implementation of FFT Algorithm in FPGA. A Digital Signal Processor with specific instruction sets and meant for a specific application is called as Application Specific Instruction set Processor (**ASIP**). An ASIP is widely used as a System on a Chip Component. Application Description Languages (ADLs) are nowadays becoming popular because of its quick and optimal design convergence achievement capability during the design of ASIPs. The first stage of designing a processor is Architecture Design Implementation.

LISA (Language For Instruction Set Architecture) is the ADL which has been used here. The platform used for design is CoWare, which allows processor architecture to be defined at an abstract level . In a similar approach to implement the processor in Hardware Description Language(here we have used VHDL), we have to make use of floating point arithmetic. This has been achieved with the help of IP cores from IP Core Generator in Xilinx ISE 10.1.

Discrete Fourier Transform is of much importance in fields of signal processing. A dedicated hardware for the frequency domain analysis of physical signals has become necessary in a large part of the electronics industry. FFT (Fast Fourier Transform) is the method of efficient calculation of DFT of a signal. It has an improved computational efficiency with respect to space and time complexity. We have implemented the 8 point radix 2 and 16 point radix 4 Cooley-Tukey algorithm in VHDL.

Contents

List of Tables

List of Figures

CHAPTER 1
INTRODUCTION

CHAPTER 1

INTRODUCTION

Today's communication market faces strong competition and multiple new standards. For this reason, several systems require new embedded processors (EP). In the current technical environment, embedded processors (EP) and the necessary development tools are designed manually, with very little automation. This results in a long, labor-intensive process requiring highly skilled engineers with specialized know-how - a very scarce resource. Most of today's processor design is conducted by EP and IC vendors using a variety of development tools from different sources, typically lacking a well-integrated and unified approach. Engineers design the architecture, simulate it in software, design software for the target application, and test the implementation for hardware and software integration. These EPs can either be general purpose, such as microcontrollers (µC) and digital signal processors (DSP), or application specific, using application specific instruction set processors (ASIP). The decision between flexibility and high optimization to special applications is driven by the time intensive task to develop new architectures. In fact, there is only little margin for design exploration and finding the best-in-class solution. This results from the fact that the development process of new ASIPs is separated into several development phases, such as design exploration, software tools design, system integration and design implementation[1].

The development time can be decreased significantly by employing a retargetable approach using a machine description language. The Language for Instruction Set Architectures (LISA)[1][2] was developed for the automatic generation of consistent software development tools and synthesizable HDL code.

Fourier Transform is basis of many signal processing and communication applications. It is the analysis of the signal in its frequency domain. The Fourier transform has many applications, in fact any field of physical science that uses sinusoidal signals, such as engineering, physics, applied mathematics, and chemistry, will make use of Fourier series and Fourier transforms. Most of the fields nowadays make use of digital and discrete data. Thus the determination of Fourier Transform of discrete signals is of prime importance and such a transform is called Discrete Fourier Transform (DFT). Fast Fourier Transform (FFT) is an efficient algorithm to evaluate DFT.

CHAPTER 2
LITERATURE REVIEW

CHAPTER 2

LITERATURE REVIEW

Frequency analysis of discrete signal can be conveniently performed on digital signal processors. In order to perform such an analysis one has to transform the signal from time domain to frequency domain representation. FFT is itself not a transformation but just a computational algorithm to evaluate Discrete Fourier Transform (DFT)[3].

The N-point discrete Fourier transform (DFT) X (k) of an N-point sequence x(n) is by definition:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} , \quad 0 \leq \text{k} \leq \text{N-1}$$

where the twiddle factor W_N is given by

$$W_N = e^{-j2\pi/N}$$

x(n) is the discrete time signal and X(k) is the signal in its frequency domain.

Direct computation of DFT is inefficient, primarily, because it does not exploit the symmetry and periodicity of the Phase Factor/ Twiddle Factor W_N.

Symmetry Property: $W_N^{k+N/2} = -W_N^k$

Periodicity property: $W_N^{k+N} = W_N^k$

Fast Fourier transform (FFT) algorithms are computationally efficient algorithms that exploit these properties of Twiddle factor. It computes the DFT of N number of discrete data samples in $O(N \log_2 N)$ time as opposed to $O(N^2)$ in the direct method[3].

FFT is a basic technique for digital signal processing applicable to spectrum analysis, digital filter, speech recognition, image processing and so on. While application fields of FFT are growing rapidly, the amount of data to be transformed is increasing tremendously. In order to efficiently compute FFT, various parallel algorithms and their implementation to processor arrays have been developed. Some of the algorithms are Radix-2, Radix-4, Quick Fourier Transform and Split Radix Transform.

2.1 Radix – 2 Algorithm:

The established equations for Radix 2 FFT are[3]:

$$X(2k) = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \left(\left[x(n) + x\left(n + \left(\frac{N}{2}\right)\right) \right] W_{\frac{N}{2}}^{kn} \right)$$

$$X(2k+1) = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \left(\left[x(n) - x\left(n + \left(\frac{N}{2}\right)\right) W_N^n \right] W_{\frac{N}{2}}^{kn} \right)$$

Where, $0 \leq k \leq (N/2)-1$

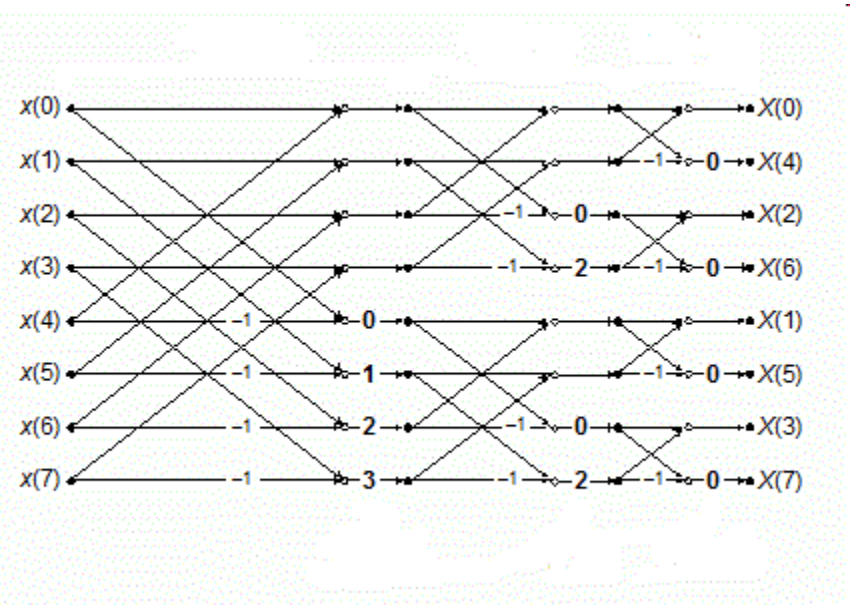

Figure 2.1 Eight Point Radix 2 DIF algorithm

The numerals at the end of each butterfly stage represents twiddle factor raised to that power.

2.2 Radix – 4 Algorithm:

The established equations for Radix 4 FFT are[3][4]:

$$X(4k) = \sum_{n=0}^{\left(\frac{N}{4}\right)-1} \left(\left[x(n) + x\left(n + \left(\frac{N}{4}\right)\right) + x\left(n + \left(\frac{N}{2}\right)\right) + x\left(n + \left(\frac{3N}{4}\right)\right) \right] W_N^0 W_{\frac{N}{4}}^{kn} \right)$$

$$X(4k+1) = \sum_{n=0}^{\left(\frac{N}{4}\right)-1} \left(\left[x(n) - j * x\left(n + \left(\frac{N}{4}\right)\right) - x\left(n + \left(\frac{N}{2}\right)\right) + j * x\left(n + \left(\frac{3N}{4}\right)\right) \right] W_N^0 W_{\frac{N}{4}}^{kn} \right)$$

$$X(4k+2) = \sum_{n=0}^{\left(\frac{N}{4}\right)-1} \left(\left[x(n) - x\left(n + \left(\frac{N}{4}\right)\right) + x\left(n + \left(\frac{N}{2}\right)\right) - x\left(n + \left(\frac{3N}{4}\right)\right) \right] W_N^0 W_{\frac{N}{4}}^{kn} \right)$$

$$X(4k+3) = \sum_{n=0}^{\left(\frac{N}{4}\right)-1} \left(\left[x(n) + j * x\left(n + \left(\frac{N}{4}\right)\right) - x\left(n + \left(\frac{N}{2}\right)\right) - j * x\left(n + \left(\frac{3N}{4}\right)\right) \right] W_N^0 W_{\frac{N}{4}}^{kn} \right)$$

Where, $0 \leq k \leq (N/4)-1$

Figure 2.2 shows the butterfly computation for radix – 4 FFT. This butterfly logic is used consecutively for calculation of N point radix – 4 DIF FFT.

Figure 2.3 shows butterfly diagram for calculation of 16 point radix – 4 DIF FFT. The numerals written after each butterfly represents multiplication of signal value with "twiddle factor raised to power numeral value".

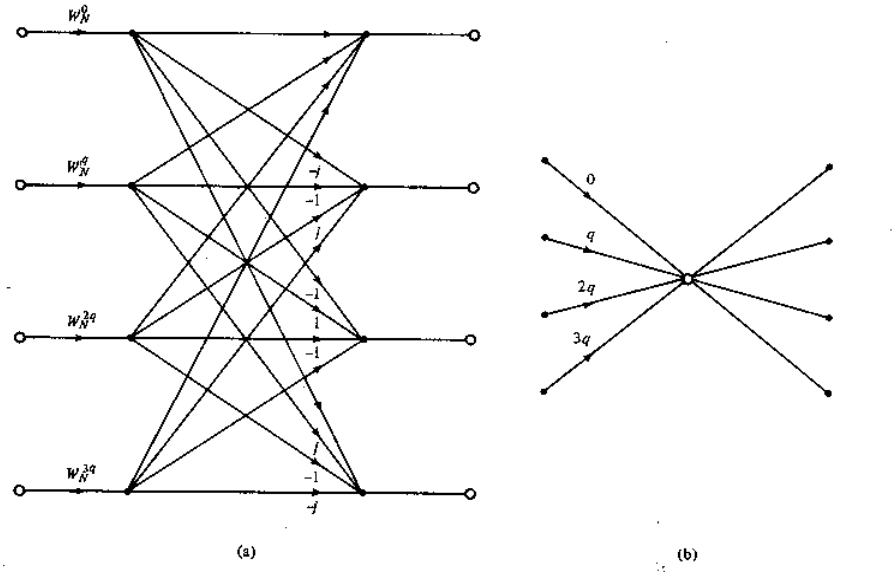

Figure 2.2 Basic Butterfly Computations in Radix-4

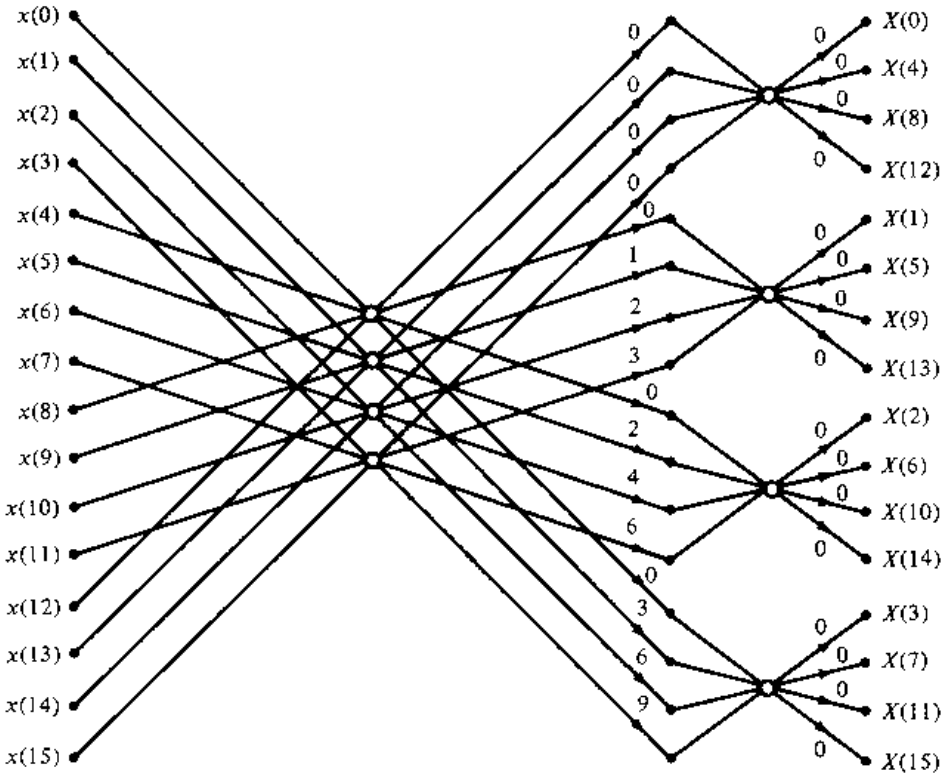

Figure 2.3 Sixteen Point Radix-4 DIF algorithm

CHAPTER 3
OVERVIEW OF LISA

CHAPTER 3

OVERVIEW OF LISA

The scope of LISA is perfectly reflected by the meaning of its acronym, that is, "Language for Instruction-Set Architectures". LISA is suited to model any architecture that is driven by an instruction set. Within this scope, LISA offers full flexibility. The language elements are generic enough to cover any kind of target architectures like GP processors, RISC processors, DSPs, ASIPs, special purpose co-processors, and so on. The instruction resource is often a register that is referred as the instruction register[5].

3.1 Advantages of using LISA

- Description of processor architecture from a higher level of abstraction other than RTL level.
- Automatic generation of all required software tools – Compiler, Assembler, and Linker.
- Automatic generation of a RTL model.

3.2 ISS Design versus Processor Design

The two mainly used models are processor design and instruction set simulator (ISS) design. ISS design is the model that typically fits the intention of all those whose main interest is an abstract model of the processor that allows to simulate its instruction set at very high speed. From such a software-centric point of view, architectural details of the processor are of little interest, while the simulation speed in terms of instructions per second is the main metric for the model quality. Here, the LISA language has basically the function of an instruction set simulation language[5][6].

On the contrary, processor design is the use model for all who actually intend to design a processor, or any feature around the processor that needs to be aware of the processor architecture. For example, consider a compiler. From such a hardware-centric point of view, the LISA model must contain all architectural details that describe the bit-accurate and cycle-accurate behavior of the processor. Here, the LISA language takes on the function of an architecture description language, though at a higher level of abstraction that conventional hardware description languages such as VHDL or Verilog.

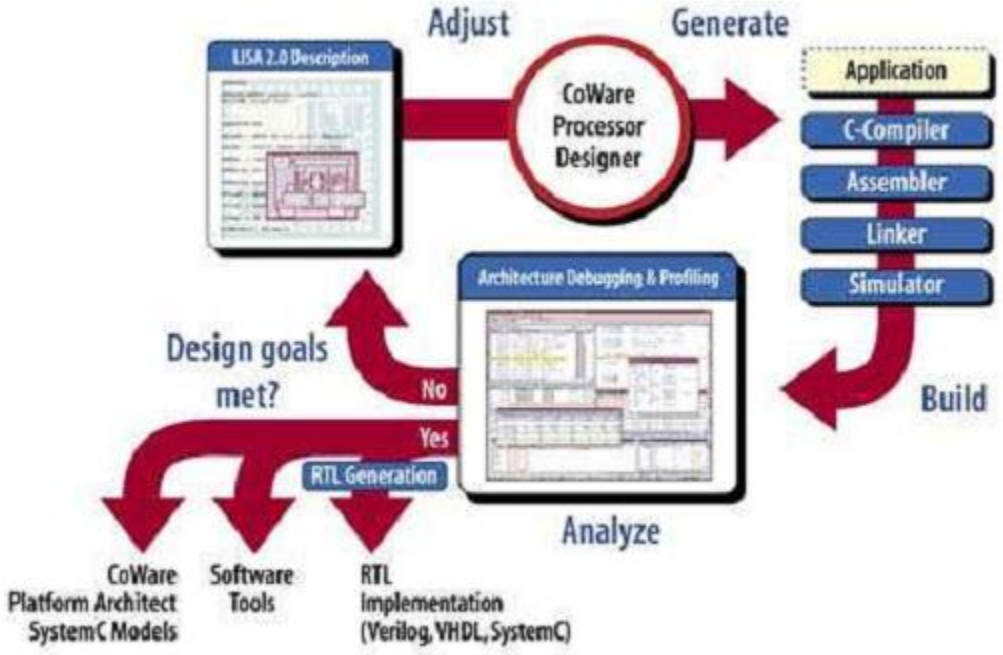

Figure 3.1 CoWare Processor Designer[7]

3.2.1 Instruction-Accurate Versus Cycle-Accurate Modeling

The ISS design and processor design use models differ mainly in the level of abstraction, or the level of accuracy, of the LISA description. Instructions set simulators have no notion of a pipeline. In every simulation control step, the complete behavior of an instruction is executed

instantly. In ISS design, the required accuracy is thus one instruction per simulation control step. Therefore, the term instruction-accurate modeling is used as a synonym for ISS design.

In pipelined architectures, a single instruction is executed in the span of multiple clock cycles, and multiple instructions are simultaneously active. Processor design requires a cycle-accurate description of the processor architecture in order to reflect the effects of the pipeline. Therefore, the term cycle-accurate modeling is used as a synonym for ISS design. At this level of abstraction, a simulation control step comprises a single clock cycle that represents a pipeline shift, rather than a complete instruction.

3.3 Building A LISA Model

Generally a processor model written in LISA has two sections those are Resource and Operation section.

3.3.1 Resources

Processor resources include the internal storage elements of the processor as well as dedicated input/output pins and global variables. The internal storage elements of the processor are represented by its registers and its internal memories. In cycle-accurate models there are other types of processor resources, like pipeline registers and interconnect signals. Processor resources are declared in the resource section, which is indicated by the keyword RESOURCE, followed by the section body limited by braces[5]. A resource declaration typically consists of an identifier, a data type specifier, and an optional keyword defining the semantic type of the resource. All resources that are declared in a resource section are global to the entire LISA model.

3.3.2 Operations:

An operation is the basic building block that describes the state transitions of the processor, namely the instructions. Operations are declared by the keyword OPERATION, followed by a unique identifier and a body limited by braces.

OPERATION <operation_identifier> {

BEHAVIOR {... // arbitrary C code}

SYNTAX {// sequence of syntax string elements}

CODING {... // sequence of bit fields}

}

A complete operation block consists of following section[6]:

3.3.2.1 Behavior Section

The instruction behavior is modeled in the behavior section of an operation. This section consists of the keyword BEHAVIOR, followed by a body limited by braces. This body contains, in principle, arbitrary C block code to describe the instruction behavior.

3.3.2.2 Syntax Section

The assembly syntax of an instruction is modeled in the syntax section of an operation. The syntax section consists of the keyword SYNTAX. In the simplest case, the syntax body contains a string or a sequence of strings that represent the assembly syntax. Literal strings are enclosed in double quotes.

3.3.2.3 Coding Section

The binary image or coding of an instruction is modeled in the coding section of an operation. The coding section consists of the keyword CODING. The coding body consists of a sequence of bit fields which consists of prefix "0b", followed by an arbitrary number of 0s, 1s, or Xs (don't-care). The most important property of the coding section is that it fully implies the instruction decoder. All tools of the Processor Designer, which require the generation of an instruction decoder, extract the necessary information from the coding sections of all operations in the LISA model. It is absolutely not necessary and not recommended to model any instruction decoding logic explicitly in the behavior section[5][6].

3.3.2.4 Declare Section

The approach of modeling every single instruction flat as a separate operation is inefficient in terms of design effort as well as in terms of performance of the generated code and tools. The construct of a *group* is important feature in LISA language to encapsulate and defer partial instruction properties to separate operations and thus create an operation hierarchy[5][6].

A group declaration must always be placed in the context of a declaration section. A declaration section is an optional section of a LISA operation. It is used to declare various LISA language-specific elements which are local to the operation. A group is just an example of a local element.

This section consists of the keyword DECLARE, followed by a body enclosed by braces.

A declaration section is always placed in the context of an operation as shown in the following example. *OPERATION ... {*

DECLARE {... // declarations of local LISA objects, for example, groups}}

3.3.2.5 Activation Section

The ACTIVATION section of an operation lists operations to be executed in subsequent control steps. This section finds its most important use in pipelining of instruction. The syntax of the ACTIVATION section looks as follows:

ACTIVATION '{' activation List '}'

3.4 Concept of Pipelining

In order to reflect the real-world hardware behavior of a processor, the modeling must be done at the granularity of clock cycles. That is, the model must be cycle-accurate. In LISA, cycle-accurate modeling implies the presence of a pipeline. A pipeline is an efficient architectural feature that allows distributing the functionality of an instruction across multiple clock cycles in a well-defined way. This procedure, which is called pipelining, enables the processor to execute more instructions per time unit, as the execution times of subsequent instructions are allowed to overlap[8].

Without a pipeline, the execution order of the different LISA operations would be as illustrated in the following figure.

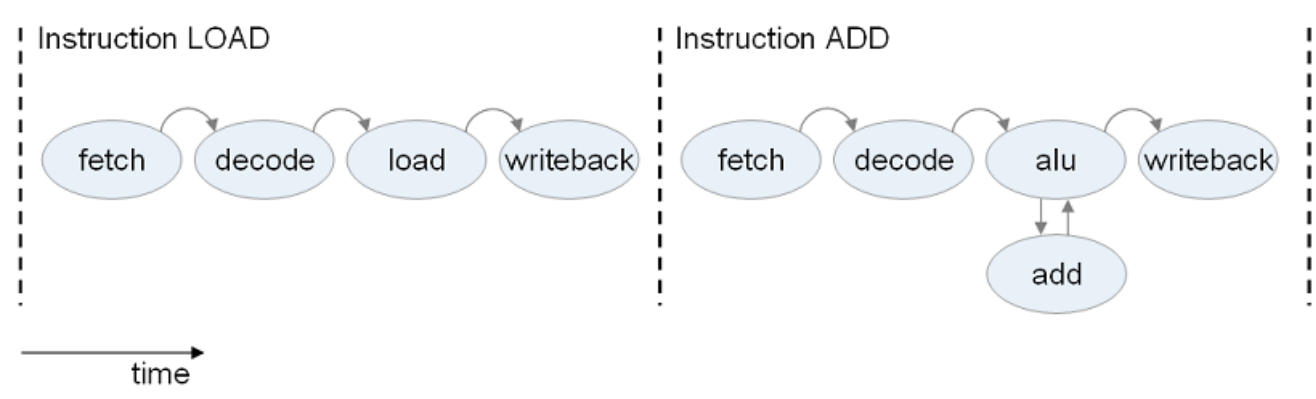

Figure 3.2 Non Pipelined Execution

Now, assume that the processor is required to execute one instruction per clock cycle. As the above figure indicates, this would require that each of the two operation sequences fetch->decode->load->writeback and fetch->decode->alu->writeback must fully fit into a single clock cycle. This requirement would typically assume either very fast but expensive hardware, or an appropriately long clock cycle and slow execution of the application on the processor.

In order to overcome this problem, one would wish to overlap the execution of the instructions in the processor. This means that the fetch functionality of the second instruction is executed immediately after the fetch functionality of the first instruction, while the first instruction continues to be executed. Such an overlapping schedule is displayed in the following figure.

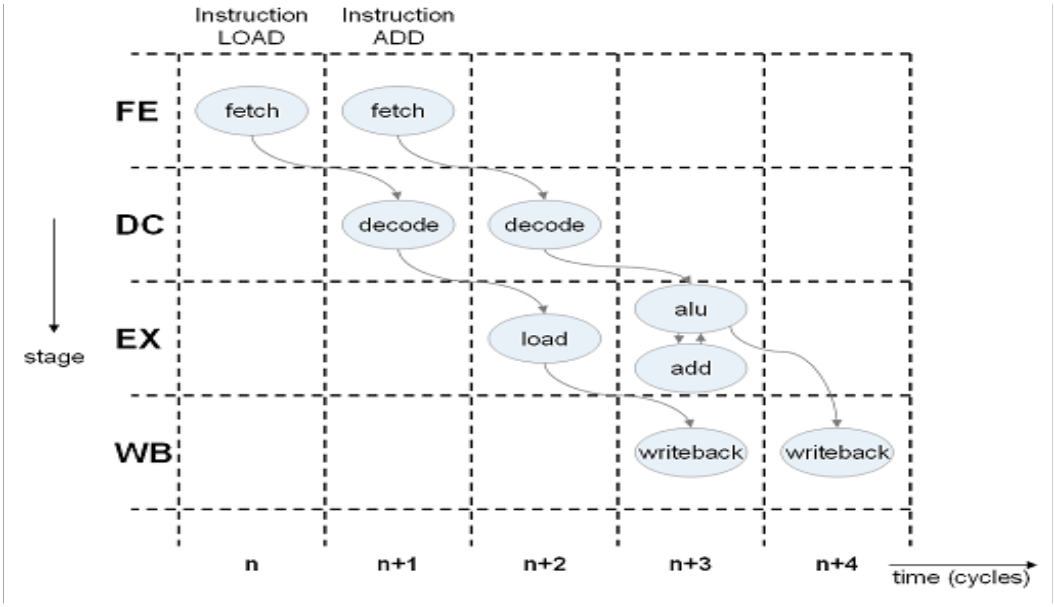

Figure 3.3 Pipelined Execution

The functionality of each instruction is distributed over four stages, which are executed in consecutive clock cycles. The entirety of these stages is called a pipeline, and the stages themselves are called pipeline stages. It is required that intermediate computational results are

stored in registers at the end of each pipeline stage. Under this condition, the minimum required duration of the clock cycle is reduced to the maximum duration of any operation sequence within as single pipeline stage, rather than the entire operation sequence of an instruction. A pipeline with ‘n’ stages allows up to ‘n’ instructions to be processed simultaneously.

Steps that are required to implement a pipeline in a LISA model include:

- defining a pipeline
- partitioning the instruction behavior across the pipeline stages
- defining the causal relations between operations in different pipeline stages
- refining the simulation control

Pipeline is declared within a resource section, using the keyword PIPELINE. As shown in the syntax below, the keyword PIPELINE is followed by an identifier, the assignment operator, and a list of pipeline stage identifiers. This list is enclosed by braces and uses a semicolon as list separator. The list is evaluated from left to right, that is, the leftmost element in the list represents the first stage in the pipeline.

RESOURCE {

PIPELINE <pipeline_identifier>{ <stage1_identifier> ; <stage2_identifier> ; ... };

}

Although it is declared in a resource section, the LISA object of a pipeline does not represent a physical resource in the hardware. It just carries the information of the number, names, and order of the pipeline stages.

3.4.1 Concept of Activation

Implementation of pipelining requires the following points to be fulfilled:

- On one hand, the causal relationship (the hierarchy) between the parent operation and the child operation must be maintained (like in the behavior call).
- On the other hand, the execution of the child operation must be delayed to the appropriate pipeline stage.

For such purposes, the LISA language provides the construct of a so-called activation section. An activation section can be used to schedule subordinate operations (that is, instances or groups) for execution, rather than executing them immediately.

An activation section is local to an operation. It consists of the keyword ACTIVATION, followed by a body limited by braces. The body contains a comma separated list of identifiers representing subordinate operations. These identifiers can be either instances, groups, or references which are declared in the same operation. The syntax of single and multiple activations is shown in the following code excerpt.

ACTIVATION { <instance_or_group_or_reference_identifier> }

ACTIVATION {

<instance_or_group_or_reference_identifier_1>,<instance_or_group_or_reference_identifier_2>

, ... }

When an activation section contains multiple elements, their execution order depends only on their location in the pipeline.

CHAPTER 4
OVERVIEW ON VHDL

CHAPTER 4

OVERVIEW ON VHDL

VHDL is a hardware description language. It is generally used to model a digital system. The digital system may be a simple logic gate or it may be a complete electronic system. A hardware abstraction of similar digital system is called an *entity*[9]. An entity 'A', when used in another entity 'B', becomes a *component* for the entity 'B'. Therefore, every component is an entity, depending on the level at which we are trying to model[9].

4.1 Design Units:

VHDL provides five types of primary constructs to design an entity. These are called "**design *units***". They are as follow:

4.1.1 Entity declaration:

An entity is modeled using an architecture body and entity declaration. The entity declaration specifies the name and lists the set of input - output ports of the entity being modeled. Ports are signals through which the entity communicates with the other components and models in its external environment[9].

4.1.2 Architecture body:

The architecture body contains the internal description of the entity[9]. It can be done by any of the following methods:

4.1.2.1 Structural style of modeling:

In this type of modeling, an entity is described as collection of interconnected components. All the used components needs to be declared in the architecture section and then they need to be port mapped individually.

4.2.1.2 Dataflow style of modeling:

In this modeling style, the flow of data through the entity is expressed using concurrent signal assignment operations. The structure of the entity cannot be visually known. But it can always be implicitly deduced.

4.2.1.3 Behavioral style of modeling:

The behavior of an entity in such type of modeling style is done as a set of statements that are executed sequentially in a specified order. These set of sequential statements are specified inside a process statement and they do not explicitly specify the structure of the entity but merely specifies its functionality. A process statement is a concurrent statement that can appear inside an architecture body.

4.2.1.4 Mixed style of modeling:

It is possible to mix all the above three discussed modeling style in a single architecture body. That is, we could use component instantiation statements (that represent structure), process statements (that represent behavior) and concurrent signal assignment statements (that represent dataflow) within an architecture body,

4.1.3 Configuration declaration:

A configuration declaration is used for creating a configuration for an entity. It specifies the binding of one architecture body from various architecture bodies that might be associated with the entity.

4.1.4 Package declaration:

A package declaration encapsulates various related declarations which can be subtype declarations, type declarations, and subprogram declaration that can be shared across multiple design units. Whenever we use a package in our VHDL code, we have to add the package using library keyword in the top of our VHDL code.

4.1.5 Package body:

A package body is used to store the definitions of procedures and functions that were declared in the corresponding package declaration. It is also used to store definitions of complete constant declarations for deferred constants that might appear in the package declaration. The package body name and the name of the package declaration with which it is associated should be exactly same. If we don't have any functions or procedures declared, then we can leave this section blank.

In order to determine the FFT of a signal, we will need to implement floating point arithmetic operations. Xilinx ISE 10.1 offers various datatypes such as real, integer, etc. **Real** datatype can be used for the floating point arithmetic operations with few limitations. Real datatype cannot be synthesized and thus, it cannot be realized on FPGA. We can only get the simulation result. Thus, the use of real datatype is to be avoided. An alternative solution to the above problem is the use of Floating Point IP Cores using IP Core generator.

4.2 Xilinx Floating Point IP Core

The Xilinx Floating-Point core allows various floating-point arithmetic operations to be performed on FPGAs. The operation to be performed by the core is specified during the generation of the core and each variant has a common interface. If a user selects an operation requiring only one operand, the B input gets omitted[10].

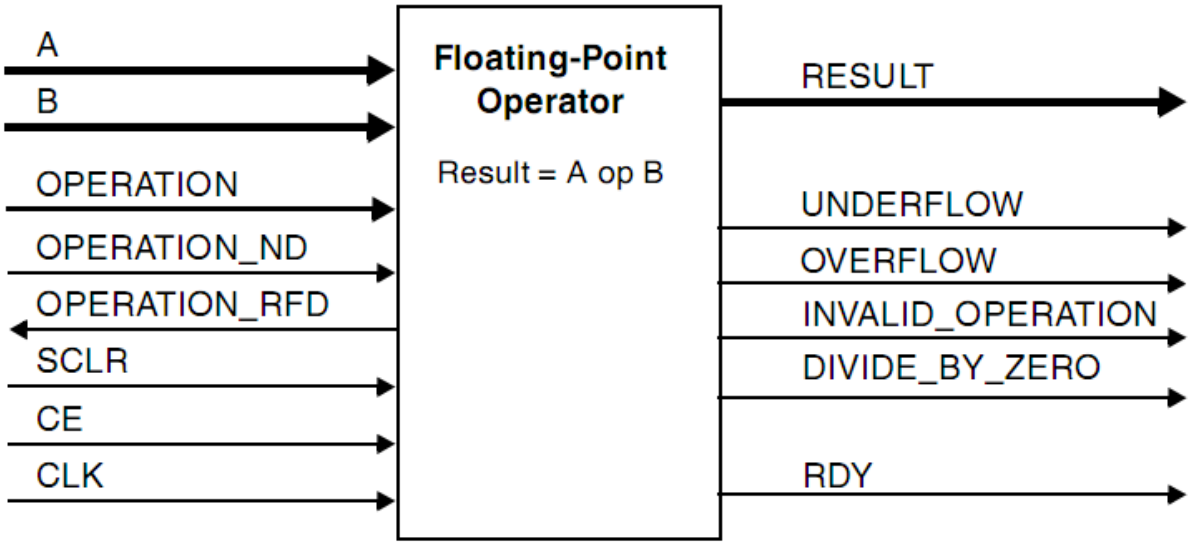

Figure 4.1 Block Diagram of Floating Point v3.0

The general floating-point representation employed in the core is the IEEE-754.

Value of a floating-point number is given by: $v = (-1)^s 2^E b_0 . b_1 b_2 \ldots b_{w_f - 1}$

The binary bits, b_i, have weighting 2^{-i}, where the most significant bit b_0 is a constant 1. If the combination is bounded such that $1 < b_0 . b_1 b_2 \ldots b_{p-1} < 2$ then, the number is said to be normalized. This quantity is scaled by a positive or negative power of 2 (denoted here as E) to provide increased dynamic range. The sign bit provides a value that is positive when s=0, and negative when s=1.

The binary representation of a floating-point number contains three fields as shown in Figure below:

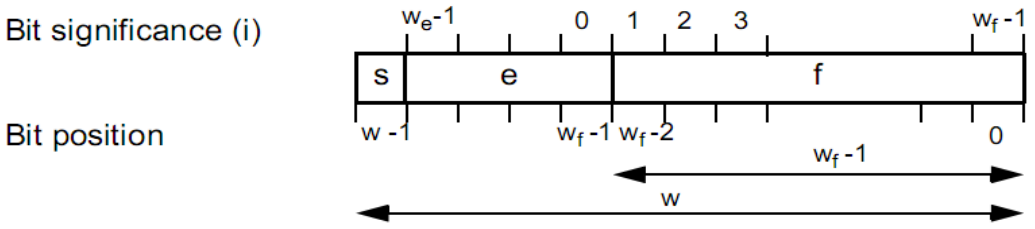

Figure 4.2 Bit Fields Within the Floating-Point Representation[10]

As b_0 is a constant, only the fractional part is retained, that is, f $=b_1 b_2 \ldots b_{w_f-1}$. This requires w_f-1 bits. From the remaining bits, $w_e = w - w_f$ bits represent the exponent and one bit is used to represent the sign.

The exponent field, e, employs a biased unsigned integer representation, whose value is given by:

$$e = \sum_{i=0}^{w_e-1} e_i 2^i$$

The index, i, of each bit within the exponent field is given in figure above. The value of the exponent, E, is obtained by removing the bias, that is,

$$E = e - (2^{w_e-1} - 1).$$

4.2.1 IEEE 754 standard

The IEEE 754 floating–point standard uses 32 bits to represent a floating-point number, including 23 bits for the significand ,1 sign bit, and 8 exponent bits. As the implied base is 2, an implied 1 is used. It means that the significand has effectively 24 bits including 1 implied bit to the left of the decimal point which is not explicitly represented in the notation.

The bias for the 8 – bit exponent is 12710 = 011111112

Table 4.3 Eight bit exponent field in IEEE 754 standard

Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
For infinities		11111111	255
127	01111111	11111110	254
...	...	...	...
2	00000010	10000001	129
1	00000001	10000000	128
0	00000000	01111111	127
-1	11111111	01111110	126
-2	11111110	01111101	125
...	...	...	...
-126	10000010	00000001	1
For Denorms	10000001	00000000	0

Briefly, the important notes on IEEE 754 are as follow,

- w = 32, w_e= 8, w_f = 24, one sign bit.
- The exponent 11111111(with all zero significand) is reserved to represent infinities or not – a – number(NAN) which may occur when a number is divided by zero.
- The smallest exponent 00000000 is reserved to represent denormalized numbers(smaller than 2^{-126} which cannot be normalized) and zero.

CHAPTER 5
RESULTS & SIMULATIONS

CHAPTER 5

RESULTS AND SIMULATIONS

The simulations were carried out in two platforms:

1) CoWare

2) Xilinx

The codes written in LISA were simulated in CoWare Processor Debugger. The first step in this regard was to generate **Assembler, Disassembler and Linker** for the LISA code. The assembly code written initially for any task is given as input to the assembler. The output of assembler is a '.lof' file which is given to the input of linker which results in creation of a '.out' file. This file is loaded in the Processor Debugger and the functionality of the instructions in the assembly code was tested. The following table shows various instructions with their binary coding and behavior.

Notations Used in the Table 5.1

GPR[]= General Purpose Register

src = source e.g. r1,r2

dest = destination e.g. r1,r2

addr= address (16 bits)

addr_value = address value (12 bits)

imm = immediate value(16 bits)

data_mem[] = data memory

Table 5.1 Instruction Set

Sl. No.	Mnemonic	Syntax	Coding	Behavior
1	nop	nop	0b0[32]	no operation
2	incr	incr src	0b0[6] src 0b0[15] 000110	GPR[src]++
3	decr	decr src	0b0[6] src 0b0[15] 000111	GPR[src]--
4	add	add dest, src1, src2	0b0[6] src1 src2 dest 0b0[5] 000001	GPR[dest]=GPR[src1]+ GPR[src2]
5	sub	sub dest, src1, src2	0b0[6] src1 src2 dest 0b0[5] 000010	GPR[dest]=GPR[src1]- GPR[src2]
6	and	and dest, src1, src2	0b0[6] src1 src2 dest 0b0[5] 000011	GPR[dest]=GPR[src1]& GPR[src2]
7	or	or dest, src1, src2	0b0[6] src1 src2 dest 0b0[5] 000100	GPR[dest]=GPR[src1] || GPR[src2]
8	mul	mul dest, src1, src2	0b0[6] src1 src2 dest 0b0[5] 000110	GPR[dest]=GPR[src1]* GPR[src2]
9	mac	mac dest,src1,src2	0b0[6] src1 src2 dest 0b0[5] 000111	GPR[dest]+=GPR[src1]* GPR[src2]
10	load	load dest, imm	0b100001 0b0[5] dest imm =obX[16]	GPR[dest]=sign_extend_16(imm)
11	ldm	ldm imm_addr,imm_value	0b1101 imm_addr=0bX[12] imm_value=0bX[16]	data_mem[imm_addr]= sign_extend_16(imm_value)
12	jmp	jmp label	0b100010 0b0[10] addr= obX[16]	Sets the PC to address of the label
13	mvm	mvm [dest+addr_value], src	0b100011 src dest 0b0[4] addr_value= obX[12]	data_mem[GPR[dest]+ addr_value]= GPR[src]
14	mov	mov dest,[src]	0b100100 src dest 0b0[16]	GPR[dest]= data_mem[GPR[src]]

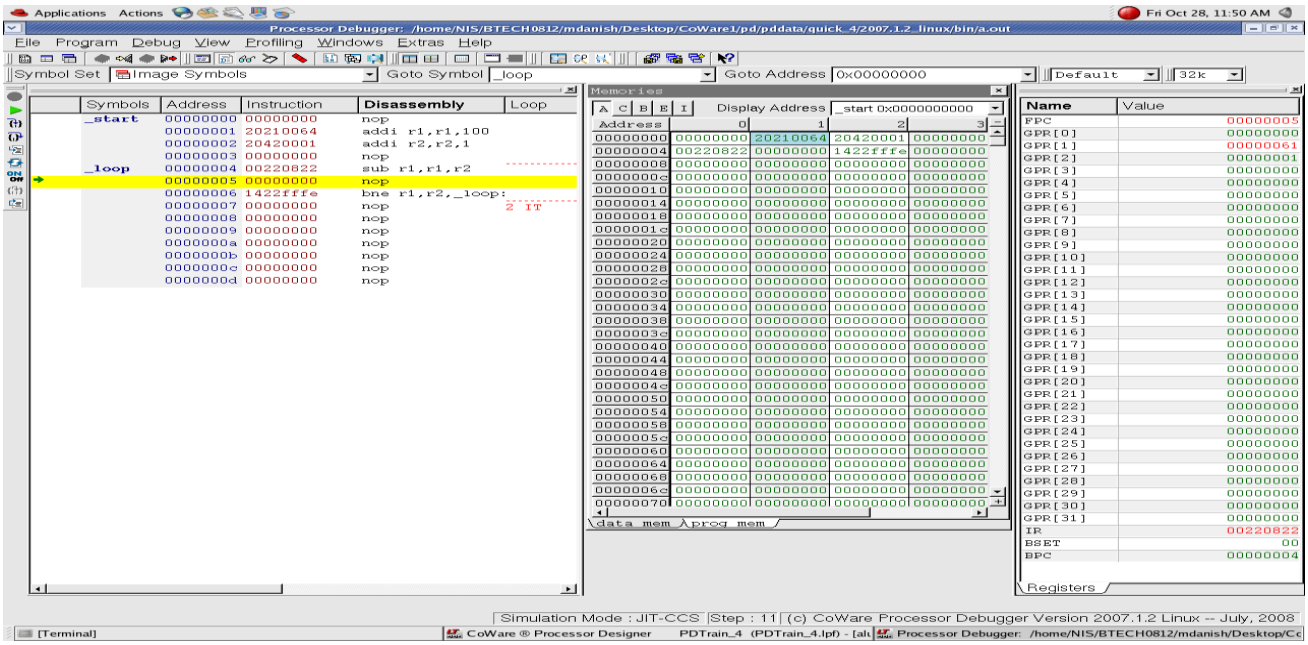

Figure 5.1 Simulation for ALU without pipeline

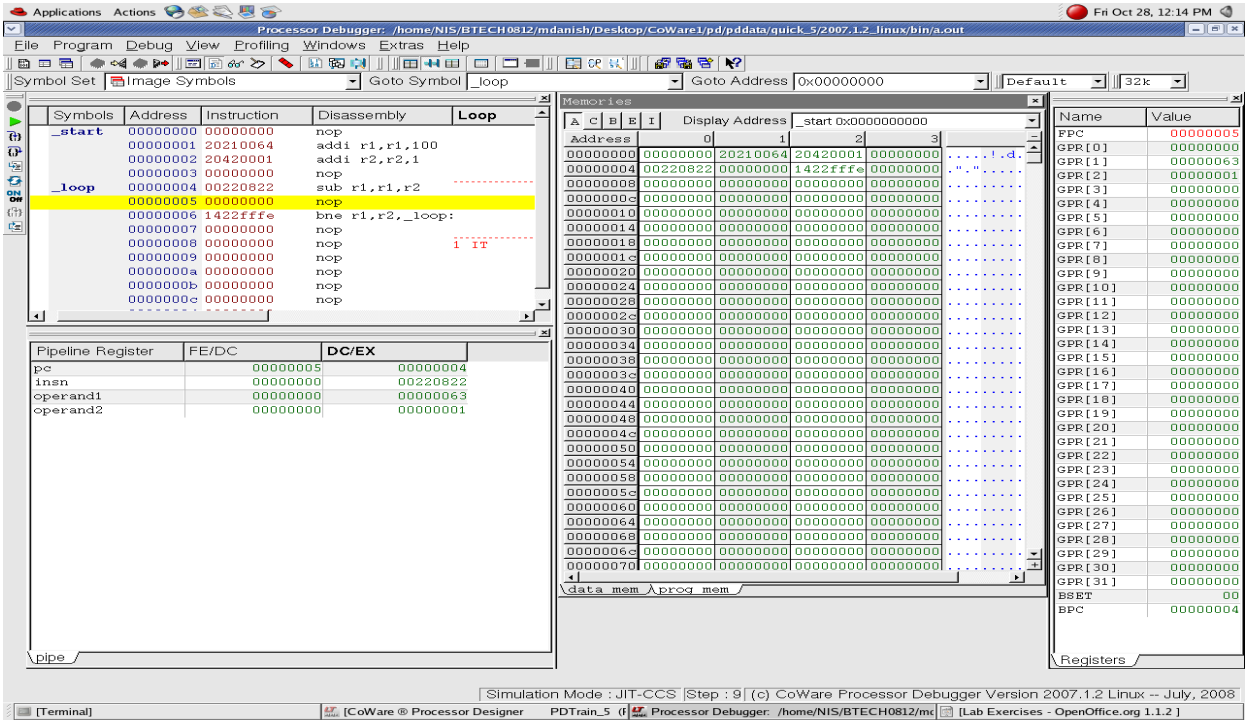

Figure 5.2 Simulation for ALU with pipeline

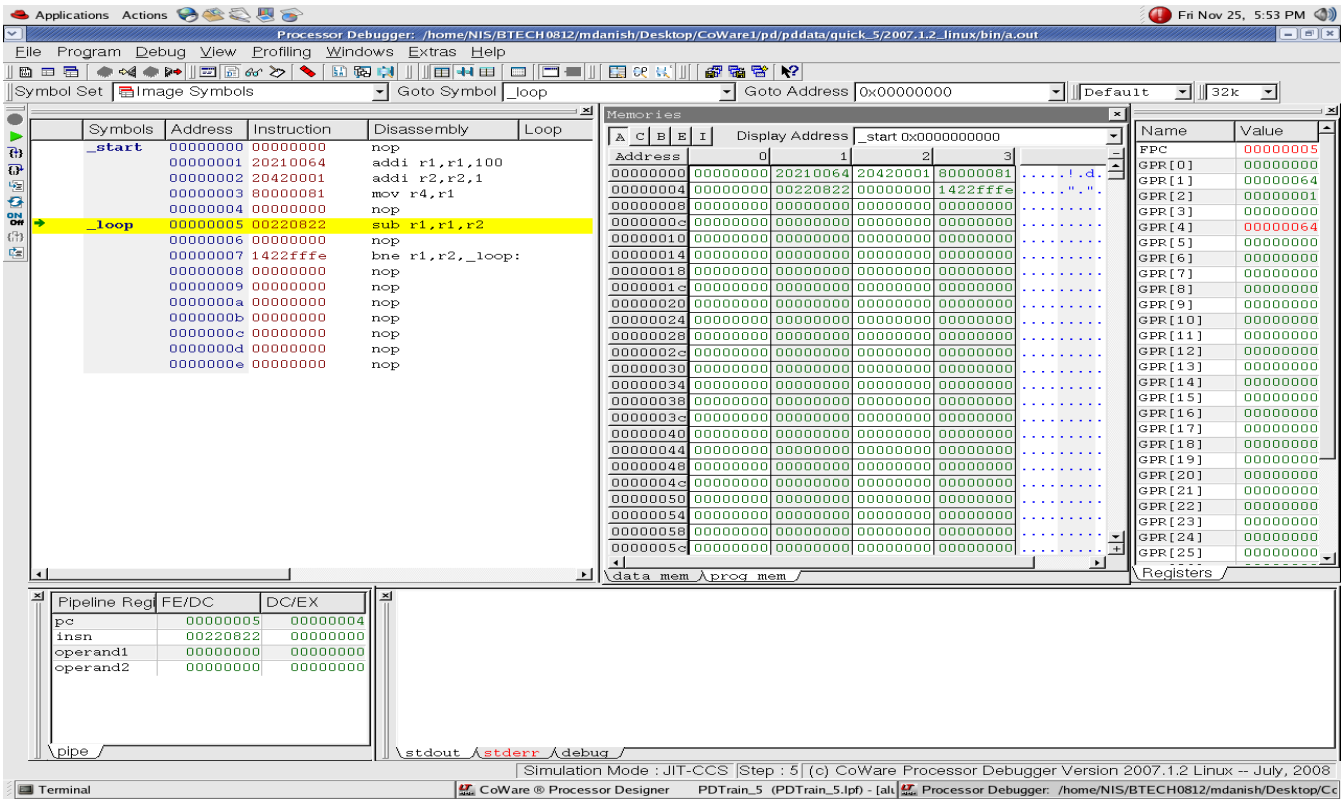

Figure 5.3 Simulation for Mov with pipeline

The simulations above show the manipulations of data in GPRs and data memory according to the assembly code. These simulations show implementation of ALU instructions, memory manipulation instructions and conditional and unconditional branching operations.

In case of the VHDL implementation we have written codes for fixed point FFT using radix 2 and radix 4 algorithm. Radix 2 was implemented for 8 points whereas radix 4 was implemented for 16 points.

Simulation 1. The input sequence for radix 2 was s={1,1,1,1,0,0,0,0}.

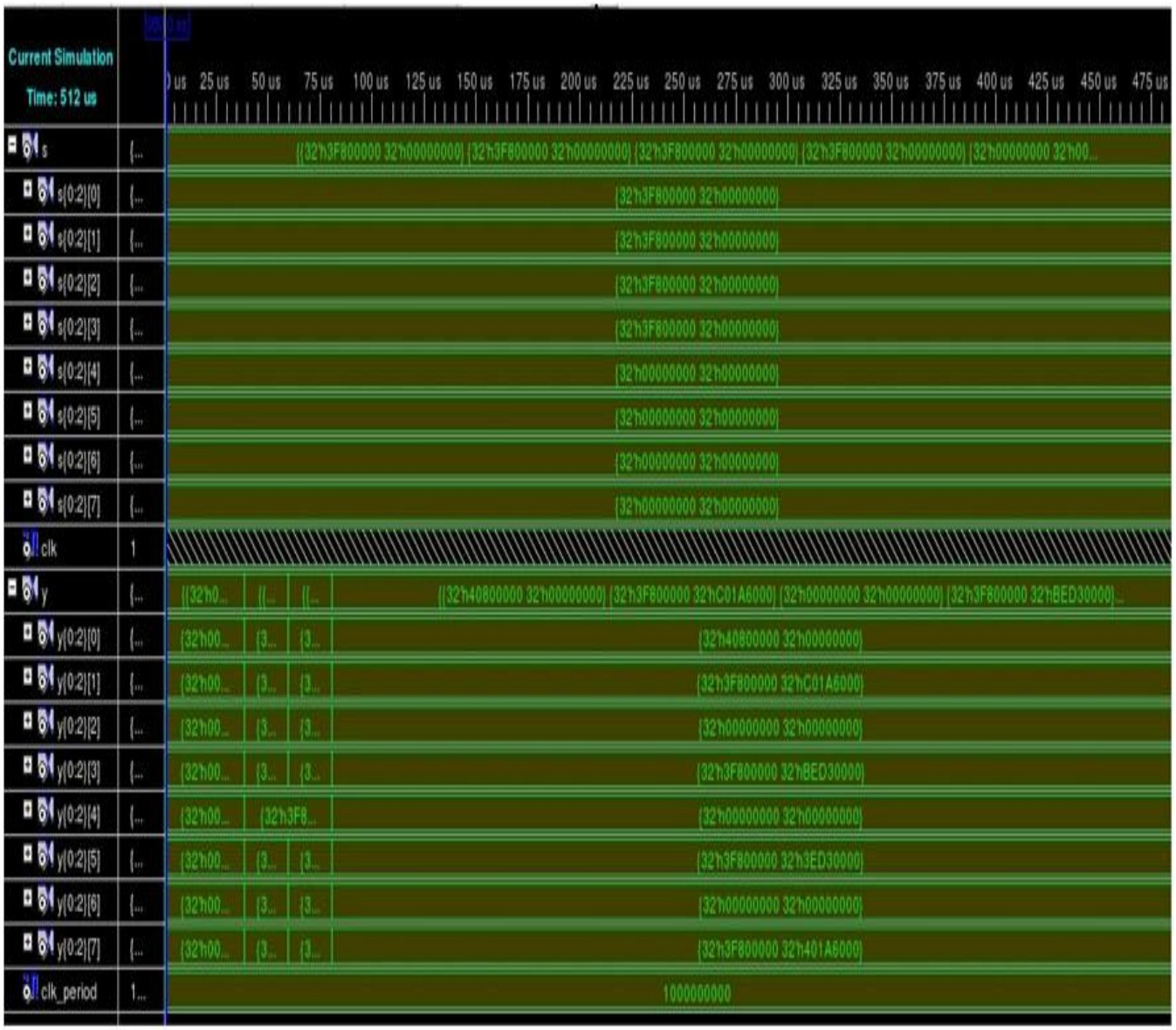

Figure 5.4 Simulation for Radix 2

Table 5.2 Radix 2 result using MATLAB and VHDL

Sl. No.	Result in MATLAB	Result from XILINX test bench (in IEEE 754 format and decimal format)
1	4 + j0	0x40800000 + j0x00000000(4 + j0)
2	1 – j2.4142	0x3f800000 + j0xc01a6000(1 – j2.412109375)
3	0 + j0	0x00000000 + j0x00000000(0 + j0)
4	1 – j0.4142	0x3f800000 + j0xbed30000(1 – j0.412109375)
5	0 + j0	0x00000000 + j0x00000000(0 + j0)
6	1 + j0.4142	0x3f800000 + j0x3ed30000(1 + j0.412109375)
7	0 + j0	0x00000000 + j0x00000000(0 + j0)
8	1 + j2.4142	0x3f800000 + j0x401a6000(1 + j2.412109375)

Simulation 2. The input sequence used is s={1,1,1,1,1,1,1,1,0,0,0,0, 0,0,0,0}.

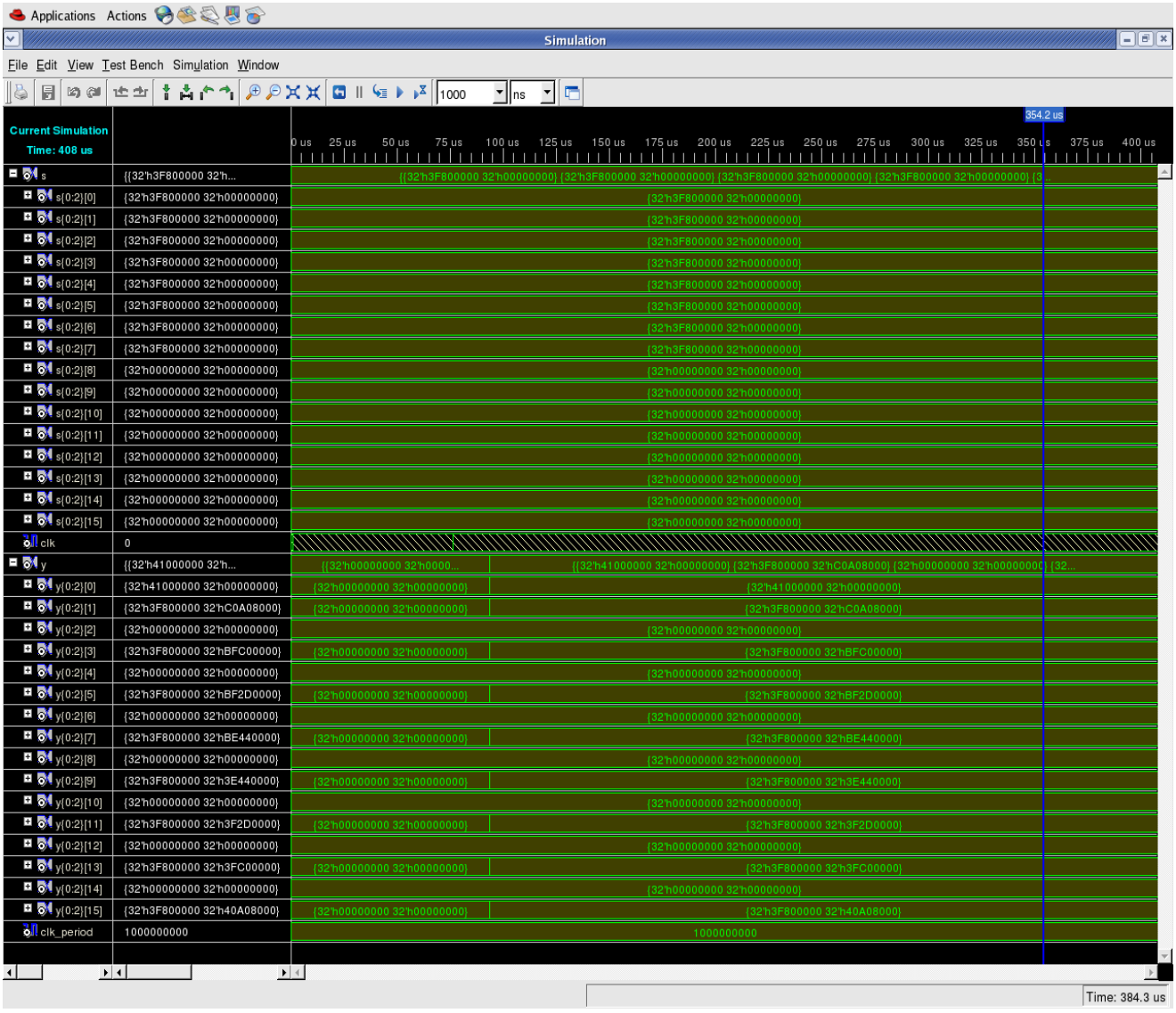

Figure 5.5 Simulation for Radix 4 using 8 butterfly blocks

Simulation 3: The input sequence used is s={1,1,1,1,1,1,1,1,0,0,0,0, 0,0,0,0}.

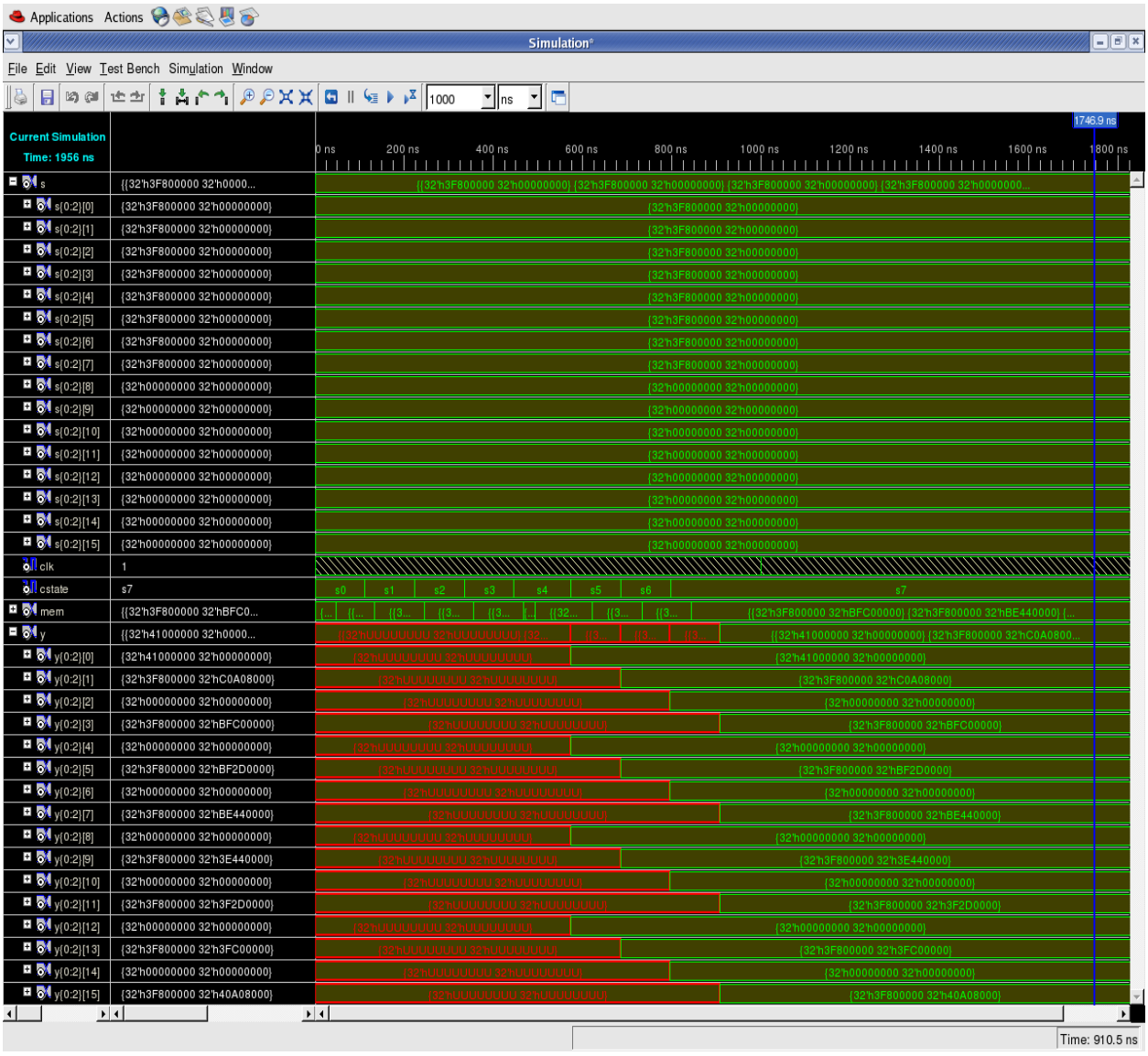

Figure 5.6 Simulation for Radix 4 using single butterfly block

Table 5.3 Radix 4 result using MATLAB and VHDL

Sl. No.	Result in MATLAB	Result from XILINX test bench (in IEEE 754 format and decimal format)
1	8 + j0	0x41000000 + j0x00000000(8 + j0)
2	1 – j5.0273	0x3f800000 + j0xc0a08000(1 – j5.015625)
3	0 + j0	0x00000000 + j0x00000000(0 + j0)
4	1 – j1.4966	0x3f800000 + j0xbfc00000(1 – j1.5)
5	0 + j0	0x00000000 + j0x00000000(0 + j0)
6	1 – j0.6682	0x3f800000 + j0xbf2d0000(1 – j0.67578125)
7	0 + j0	0x00000000 + j0x00000000(0 + j0)
8	1 – j0.1989	0x3f800000 + j0xbe440000(1 – j0.19140625)
9	0 + j0	0x00000000 + j0x00000000(0 + j0)
10	1 + j0.1989	0x3f800000 + j0x3e440000(1 + j0.19140625)
11	0 + j0	0x00000000 + j0x00000000(0 + j0)
12	1 + j0.6682	0x3f800000 + j0x3f2d0000(1 + j0.67578125)
13	0 + j0	0x00000000 + j0x00000000(0 + j0)
14	1 + j1.4966	0x3f800000 + j0x3fc00000(1 + j1.5)
15	0 + j0	0x00000000 + j0x00000000(0 + j0)
16	1 + j5.0273	0x3f800000 + j0x40a08000(1 + j5.015625)

The radix 2 code was first successfully synthesized for **xc3s500e-5fg320** (Spartan 3E family) and the results obtained were found to match from that of the MATLAB code. For implementation of radix – 4, first a butterfly component was designed, and then the component was port mapped 8 times for complete 16 point radix 4 FFT. The VHDL code was then synthesized and the simulation was compared with MATLAB result. The radix 4 code was further improved by "port mapping butterfly component once and using 8 states", instead of 8 butterfly components individually. The improved code was synthesized and simulation was observed. The design summary comparison for both radix – 4 program are shown in the following table.

Table 5.4 Design Summary comparison between both radix – 4 program in VHDL

Logic utilization	Used in radix 4 code	Used in improved radix 4 code
Number of slices	130272	17083
Number of slice flip flops	206272	26820
Number of 4 input LUTs	194368	25624
Number of bonded IOBs	2049	2049
Number of GCLKs	1	9

CHAPTER 6

CONCLUSIONS & SCOPE FOR FUTURE WORK

CHAPTER 6

CONCLUSIONS AND SCOPE FOR FUTURE WORK

The simulations in the CoWare platform indicate that operational hierarchy and pipelining have a reduced clock cycle usage compared to a flat instruction set processor. Therefore, 'Cycle Accurate model' should be used for processor design.

Fixed Point FFTs in radix 2 and radix 4 are implemented in this project. The results are validated from the input signal and twiddle factor assigned in the test bench. The outputs obtained from VHDL code and MATLAB codes are in close agreement confirming satisfactory results. The comparison between different coding techniques for radix 4 method is performed. The use of "states" in place of port mapping reduces the device resource utilization to a great extent. Hence, similar improvement in code can lead to a large variation in resource utilization.

We have demonstrated the improvement of top module of the VHDL code for radix 4 using states. Design need to be further optimized to reduce the area constraints in FPGA. A more efficient use of the device resources can be achieved by continuously improving the butterfly component till we are left with only the necessary cores i.e. core for arithmetic and multiplication operation. FFT assembly code is to be implemented using cycle accurate model in LISA. Moreover, the task of real time N point FFT calculation can be taken up as a future work. A more complex application specific assembly code using only the instructions in the table can be implemented and output can be observed in the processor debugger. In case of requirement of additional instructions, one can add *Operations* in the LISA code for describing functionality.

REFERENCES

[1] O. Schliesbusch, A. Hoffmann, A. Nohl, G. Braun and H. Meyr, ”Architecture Implementation Using the Machine Description Language LISA”,IEEE Proceedings of the 15th International Conference on VLSI Design (VLSIDí02),2002

[2] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA –“Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures”. In Proc. of the Design Automation Conference (DAC), June 1999.

[3] J.G.Proakis, D.G.Manolakis, ”DSP Principles, Algorithms and Applications”, PHI Publications,4th edition,2007.

[4] S. Bouguezel, M. Omair Ahmad and M.N.S. Swamy, “IMPROVED RADIX-4 AND RADIX-8 FFT ALGORITHMS”, IEEE ISCAS 2004.

[5] LISA Modeling Fundamentals , “CoWare Reference Manuals V2.0”

[6] LISA Language Reference, “CoWare Reference Manuals V2.0”

[7] U.Nanda, Design of an Application Specific Instruction Set Processor Using LISA, M.Tech Thesis, Dept. Electron.Comm.Eng.,N.I.T. Rourkela, Rourkela, Orissa,2010.

[8] Processor Design Guide , “CoWare Reference Manuals V2.0”

[9] J. Bhasker, ”A VHDL Primer”, PHI Publications, Third Edition, 1999

[10] User Reference Manual for Floating Point v3.0, provided by Xilinx Inc.